Semantically Correct Query Answers in the Presence of Null Values

Loreto Bravo and Leopoldo Bertossi

Carleton University, School of Computer Science
Ottawa, Canada.
{lbravo,bertossi}@scs.carleton.ca

Abstract. For several reasons a database may not satisfy a given set of integrity constraints (ICs), but most likely most of the information in it is still consistent with those ICs; and could be retrieved when queries are answered. Consistent answers to queries wrt a set of ICs have been characterized as answers that can be obtained from every possible minimally repaired consistent version of the original database. In this paper we consider databases that contain null values and are also repaired, if necessary, using null values. For this purpose, we propose first a precise semantics for IC satisfaction in a database with null values that is compatible with the way null values are treated in commercial database management systems. Next, a precise notion of repair is introduced that privileges the introduction of null values when repairing foreign key constraints, in such a way that these new values do not create an infinite cycle of new inconsistencies. Finally, we analyze how to specify this kind of repairs of a database that contains null values using disjunctive logic programs with stable model semantics.

1 Introduction

In databases, integrity constraints (ICs) capture the semantics of the application domain, and help maintain the correspondence between this domain and the database when updates are performed. However, there are several reasons for a database to be or become inconsistent wrt a given set of ICs [6]; and sometimes it could be difficult, impossible or undesirable to repair the database in order to restore consistency [6]. This process might be too expensive; useful data might be lost; it may not be clear how to restore the consistency, and sometimes even impossible, e.g. in virtual data integration, where the access to the autonomous data sources may be restricted [9].

In those situations, possibly most of the data is still consistent and can be retrieved when queries are posed to the database. In [2], consistent data is characterized as the data that is invariant under certain minimal forms of restoration of consistency, i.e. as the data that is present in all minimally repaired and consistent versions of the original instance, the so-called *repairs*. In particular, an answer to a query is defined as consistent when it can be obtained as a standard answer to the query from *every possible* repair.

More precisely, a repair of a database instance D, as introduced in [2], is a new instance of the same schema as D that satisfies the given ICs, and makes minimal under set inclusion the symmetric set difference with the original instance, taken both instances as sets of ground database atoms.

In [2, 13, 14, 17] algorithms and implementations for consistent query answering (CQA) have been presented, i.e. for retrieving consistent answers from inconsistent databases. All of them work only with the original, inconsistent

database, without restoring its consistency. That is, inconsistencies are solved at query time. This is in correspondence with the idea that the above mentioned repairs provide an auxiliary concept for defining the right semantics for consistent query answers. However, those algorithms apply to restricted classes of queries and constraints, basically those for which the intrinsic complexity of CQA is still manageable [15].

In [3, 20, 5, 6] a different approach is taken: database repairs are specified as the stable models of disjunctive logic programs, and in consequence consistent query answering amounts to doing *cautious* or *certain* reasoning from logic programs under the stable model semantics. In this way, it is possible to handle any set of universal ICs and any first-order query, and even beyond that, e.g. queries expressed in extensions of Datalog. It is important to realize that the data complexity of query evaluation in disjunctive logic programs with stable model semantics [16] matches the intrinsic data complexity of CQA [15], namely both of them are Π_2^P-complete.

All the previous work cited before did not consider the possible presence of null values in the database, and even less their peculiar semantics. Using null values to repair ICs was only slightly considered in [3, 5, 6]. This strategy to deal with referential ICs seemed to be the right way to proceed given the results presented in [11] that show that repairing cyclic sets of referential ICs by introducing arbitrary values from the underlying database domain leads to the undecidability of CQA.

In [10] the methodology presented in [5, 6], based on specifying repairs using logic programs withe extra annotation constants, was systematically extended in order to handle both; (a) databases containing null values, and (b) referential integrity constraints (RICs) whose satisfaction is restored via introduction of null values. According to the notion of IC satisfaction implicit in [10], those introduced null values do not generate any new inconsistencies.

Here, we extend the approach and results in [10] in several ways. First, we give a precise semantics for integrity constraint satisfaction in the presence of null values that is both sensitive to *the relevance of the occurrence of a null value* in a relation, and also compatible with the way null values are usually treated in commercial database management systems (the one given in [10] was much more restrictive). The introduced null values do not generate infinite repair cycles through the same or other ICs, which requires a semantics for integrity constraint satisfaction under null values that sanctions that tuples with null values in attributes relevant for checking the IC do not generate any new inconsistencies. A new notion of repair is given accordingly. With the new repair semantics CQA becomes decidable for a quite general class of ICs that includes universal constraints, referential ICs, *NOT NULL*-constraints, and foreign key constraints, even the cyclic cases.

The logic programs that specify the repairs are modified wrt those given in [10], in such a way that the expected one-to-one correspondence between the stable models and repairs is recovered for *acyclic* sets of RICs. Finally, we study classes of ICs for which the specification can be optimized and a lower complexity for CQA can be obtained.

2   Preliminaries

We concentrate on relational databases, and we assume we have a fixed relational schema $\Sigma = (\mathcal{U}, \mathcal{R}, \mathcal{B})$, where $\mathcal{U}$ is the possibly infinite database domain such

that $null \in \mathcal{U}$, $\mathcal{R}$ is a fixed set of database predicates, each of them with a finite, ordered set of attributes, and $\mathcal{B}$ is a fixed set of built-in predicates, like comparison predicates. $R[i]$ denotes the attribute in position i of predicate $R \in \mathcal{R}$. The schema determines a language $\mathcal{L}(\Sigma)$ of first-order predicate logic. A database instance D compatible with Σ can be seen as a finite collection of ground atoms of the form $R(c_1, ..., c_n)$,[1] where R is a predicate in $\mathcal{R}$ and $c_1, ..., c_n$ are constants in $\mathcal{U}$. Built-in predicates have a fixed extension in every database instance, not subject to changes. We need to define ICs because their syntax is fundamental for what follows.

An *integrity constraint* is a sentence $\psi \in \mathcal{L}(\Sigma)$ of the form:

$$\forall \bar{x}(\bigwedge_{i=1}^{m} P_i(\bar{x}_i) \longrightarrow \exists \bar{z}(\bigvee_{j=1}^{n} Q_j(\bar{y}_j, \bar{z}_j) \vee \varphi)), \tag{1}$$

where $P_i, Q_j \in \mathcal{R}$, $\bar{x} = \bigcup_{i=1}^{m} \bar{x}_i$, $\bar{z} = \bigcup_{j=1}^{n} \bar{z}_j$, $\bar{y}_j \subseteq \bar{x}$, $\bar{x} \cap \bar{z} = \emptyset$, $\bar{z}_i \cap \bar{z}_j = \emptyset$ for $i \neq j$, and $m \geq 1$. Formula φ is a disjunction of built-in atoms from $\mathcal{B}$, whose variables appear in the antecedent of the implication. We will assume that there is a propositional atom **false** $\in \mathcal{B}$ that is always false in a database. Domain constants other than *null* may appear instead of some of the variables in a constraint of the form (1). When writing ICs, we will usually leave the prefix of universal quantifiers implicit. A wide class of ICs can be accommodated in this general syntactic class by appropriate renaming of variables if necessary.

A *universal integrity constraint* (UIC) has the form (1), but with $\bar{z} = \emptyset$, i.e. without existentially quantified variables:

$$\bar{\forall} \bar{x}(\bigwedge_{i=1}^{m} P_i(\bar{x}_i) \longrightarrow \bigvee_{j=1}^{n} Q_j(\bar{y}_j) \vee \varphi). \tag{2}$$

A *referential integrity constraint* (RIC) is of the form (1), but with $m = n = 1$ and $\varphi = \emptyset$, i.e. of the form[2]: (here $\bar{x}' \subseteq \bar{x}$ and $P, Q \in \mathcal{R}$)

$$\forall \bar{x} \ (P(\bar{x}) \longrightarrow \exists \bar{y} \ Q(\bar{x}', \bar{y})). \tag{3}$$

Class (1) includes most ICs commonly found in database practice, e.g. a *denial constraint* can be expressed as $\bar{\forall} \bar{x}(\bigwedge_{i=1}^{m} P_i(\bar{x}_i) \longrightarrow$ **false**). Functional dependencies can be expressed by several implications of the form (1), each of them with a single equality in the consequent. Partial inclusion dependencies are RICs, and full inclusion dependencies are universal constraints. We can also specify (single row) *check constraints* that allow to express conditions on each row in a table, so they can be formulated with one predicate in the antecedent of (1) and only a formula φ in the consequent. For example, $\forall xy(P(x, y) \rightarrow y > 0)$ is a check constraint.

In the following we will assume that we have a fixed finite set *IC* of ICs of the form (1). Notice that sets of constraints of this form are always consistent in the classical sense, because empty database always satisfy them.

Example 1. For $\mathcal{R} = \{P, R, S\}$ and $\mathcal{B} = \{>, =, \textbf{false}\}$, the following are ICs: (a) $\forall xyzw \ (P(x, y) \wedge R(y, z, w) \rightarrow S(x) \vee (z \neq 2 \vee w \leq y))$ (universal). (b)$\forall xy(P(x, y) \rightarrow \exists z \ R(x, y, z))$ (referential). (c)$\forall x(S(x) \rightarrow \exists yz(R(x, y) \vee R(x, y, z)))$. $\square$

---

[1] Also called *database tuples*. Finite sequences of constants in $\mathcal{U}$ are simply called *tuples*.

[2] To simplify the presentation, we are assuming the existential variables appear in the last attributes of Q, but they may appear anywhere else in Q.

Notice that defining φ in (1) as a disjunction of built-in atoms is not an important restriction, because an IC that has φ as a more complex formula can be transformed into a set of constraints of the form (1). For example, the formula $\forall xy\, (P(x, y) \rightarrow (x > y \vee (x = 3 \wedge y = 8)))$ can be transformed into: $\forall xy\, (P(x, y) \rightarrow (x > y \vee x = 3))$ and $\forall xy\, (P(x, y) \rightarrow (x > y \vee y = 8))$.

The *dependency graph* $\mathcal{G}(IC)$ [12] for a set of ICs IC of the form (1) is defined as follows: Each database predicate P in $\mathcal{R}$ appearing in IC is a vertex, and there is a directed edge (P_i, P_j) from P_i to P_j iff there exists a constraint $ic \in IC$ such that P_i appears in the antecedent of ic and P_j appears in the consequent of ic.

Example 2. For the set IC containing the UICs $ic_1 : S(x) \rightarrow Q(x)$ and $ic_2 : Q(x) \rightarrow R(x)$, and the RIC $ic_3 : Q(x) \rightarrow \exists y T(x, y)$, the following is the dependency graph $\mathcal{G}(IC)$:
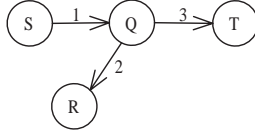

the edges are labelled just for reference. Edges 1 and 2 correspond to the constraints ic_1 and ic_2, resp., and edge 3 to ic_3. □

A *connected component* in a graph is a maximal subgraph such that for every pair (A, B) of its vertices, there is a path from A to B or from B to A. For a graph $\mathcal{G}$, $\mathcal{C}(\mathcal{G}) := \{c \mid c$ is a connected component in $\mathcal{G}\}$; and $\mathcal{V}(\mathcal{G})$ is the set of vertices of $\mathcal{G}$.

Definition 1. Given a set IC of UICs and RICs, IC_U denotes the set of UICs in IC. The *contracted dependency graph*, $\mathcal{G}^C(IC)$, of IC is obtained from $\mathcal{G}(IC)$ by replacing, for every $c \in \mathcal{C}(\mathcal{G}(IC_U))$,[3] the vertices in $\mathcal{V}(c)$ by a single vertex and deleting all the edges associated to the elements of IC_U. Finally, IC is said to be *RIC-acyclic* if $\mathcal{G}^C(IC)$ has no cycles. □

Example 3. (example 2 cont.) The contracted dependency graph $\mathcal{G}^C(IC)$ is obtained by replacing in $\mathcal{G}(IC)$ the edges 1 and 2 and their end vertices by a vertex labelled with $\{Q, R, S\}$.
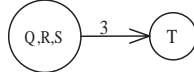
Since there are no loops in $\mathcal{G}^C(IC)$, IC is RIC-acylic. If we add a new UIC: $T(x, y) \rightarrow R(y)$ to IC, all the vertices belong to the same connected component. $\mathcal{G}(IC)$ and $\mathcal{G}^C(IC)$ are, respectively:

Since there is a self-loop in $\mathcal{G}^C(IC)$, the new IC is *not* RIC-acylic. □

As expected, a set of UICs is always RIC-acyclic.

---

[3] Notice that for every $c \in \mathcal{C}(\mathcal{G}(IC_U))$, it holds $c \in \mathcal{C}(\mathcal{G}(IC))$.

3   IC Satisfaction in Databases with Null Values

We deal with incomplete databases in the classic sense that some information is represented using null values [21] (cf. also [19]). More recently, the notion of incomplete database has been used in the context of virtual data integration [23, 9], referring to data sources that contain a subset of the data of its kind in the global system; and in inconsistent databases [11, 15], referring to the fact that inconsistencies may have occurred due to missing information and then, repairs are obtained through insertion of new tuples.

There is no agreement in the literature on the semantics of null values in relational databases. There are several different proposals in the research literature [29, 4, 25, 28], in the SQL standard [31, 22], but also implicit semantics in the different ways null values are handled in commercial database management systems (DBMSs).

Not even within the SQL standard there is a homogenous and global semantics of integrity constraint satisfaction in databases with null values; rather, different definitions of satisfaction are given for each type of constraint. Actually, in the case of foreign key constraints, three different semantics are suggested (*simple*-, *partial*- and *full-match*). Commercial DBMSs implement only the simple-match semantics for foreign key constraints. Some criticisms to the treatment of nulls in the SQL standard have been expressed by the database community, c.f. [32].

One of the reasons why it is difficult to agree on a semantics is that a null value can be interpreted as an unknown, inapplicable or even withheld value. Different null constants can be used for each of these different interpretations [27]. Also the use of more than one null value (of the same kind), i.e. labelled nulls, has been suggested [30], but in this case every new null value uses a new fresh constant; for which the *unique names assumption* does not apply. The latter alternative allows to keep a relationship between null values in different attributes or relations. However commercial DBMSs consider only one null value, represented by a single constant, that can be given any of the interpretations mentioned above.

In [10] a semantics for null values was adopted, according to which a tuple with a null value in any of its attributes would not be the cause for any inconsistencies. In other words, it would not be necessary to check tuples with null values wrt possible violations of ICs (except for *NOT NULL*- constraints, of course). This assumption is consistent in some cases with the practice of DBMSs, e.g. in IBM DB2 UDB. Here we will propose a semantics that is less liberal in relation to the participation of null values in inconsistencies; a sort of compromise solution considering the different alternatives available.

Example 4. For *IC* containing only $\psi_1 \colon P(x, y, z) \to R(y, z)$, the database $D = \{P(a, b, null)\}$ would be: (a) Consistent wrt the semantics in [10] because there is a null value in the tuple (b) Consistent wrt the simple-match semantics of SQL:2003 [22], because there is a null value in one of the attributes in the set $\{P[2], P[3], R[1], R[2]\}$ of attributes that are relevant to check the constraint. (c) Inconsistent wrt the partial-match semantics in SQL:2003, because there is no tuple in R with a value b in its first attribute. (d) Inconsistent wrt the full-match semantics in SQL:2003, because there cannot be a *null* in an attribute that is referencing a different table.

If we consider, instead of ψ_1, the constraint $\psi_2 \colon P(x, y, z) \to R(x, y)$, the same database would be consistent only for the semantics in [10], because the

other semantics consider only the null value in the attributes that are relevant to check the constraint, and in this case there is no null value there. $\square$

Even though there are different possible semantics, we would like to define and concentrate in a null-value semantics that would allow us to integrate our results with commercial databases. This is why we would like to generalize in a declarative and homogenous way the semantics defined in SQL:2003 [22] that is implemented in commercial DBMSs like IBM DB2 UDB. For this reason we consider only one kind of null value. We also want our null-value semantics to be uniform for a wide class of ICs that goes beyond the type of constraints supported by commercial DBMS.

Example 5. Consider a database with a table that stores courses with the professor that taught it and the term, and a table that stores the experience of each professor in each course with the number of times (s)he has taught the course. We have a foreign key constraint based on the RIC $\forall xyz(Course(x,y,z) \rightarrow \exists w \ Exp(y, \ x, \ w))$ together with the constraint expressing that table *Exp* has $\{ID, Code\}$ as a key. We can be sure there are no null values in those two attributes. Now consider the instance D:

Course	Code	ID	Term
	CS27	21	W04
	CS18	34	null
	CS50	null	W05

Exp	ID	Code	Times
	21	CS27	3
	34	CS18	null
	45	CS32	2

In IBM DB2, this database is accepted as consistent. The null values in columns *Term* and *Times* are not relevant to check the satisfaction of the constraints. In order to check the constraint the only attributes that we need to pay attention to are *ID* and *Code*. If *null* is in the one of these attributes in table *Course*, the tuple is considered to be consistent, without checking table *Exp*. For example *Course(CS50,null,W05)* has a null value in *ID*, therefore DB2 does not check if there is a tuple in *Exp* that satisfies the constraint. It does not even check that there exists a tuple in *Exp* with attribute *Code=CS50*.

This behavior for foreign key constraints is called simple-match in the SQL standard, and is the one implemented in all commercial DBMS. The partial- and full-match would not accept the database as consistent, because partial-match would require *Exp* to have a tuple (any non-null value,*CS50*, any value); and full-match would not allow a tuple with *null* in attributes *ID* or *Code* in table *Course*.

If we try to insert tuple *(CS41,18, null)* into table *Course*, it would be rejected by DB2. This is because the attributes *ID* and *Code* are relevant to check the constraint and are different from *null*, but there is no tuple in *Exp* with *ID=18* and *Code=CS41*. $\square$

Example 6. Consider the single-row check constraint $\forall ID \ \forall Name \ \forall Salary \ (Emp (ID, \ Name, \ Salary) \rightarrow Salary > 100)$ and the database D below. DB2 accepts

Emp	ID	Name	Salary
	32	null	1000
	41	Paul	null

this database instance as consistent. Here, in order to check the satisfaction of the constraint, we only need to verify that the attribute *Salary* is bigger than 100; therefore the only attribute that is relevant to check the constraint is *Salary*. DBMSs will accept as consistent any state where the condition (the consequent) evaluates to *true* or *unknown*.

The latter is the case here. Tuple *(32, null, 50)* could not be inserted because *Salary* > 100 evaluates to *false*. Notice that the null values in attributes other that *Salary* are not even considered in the verification of the satisfaction. □

When dealing with primary keys, DBMSs use a bag semantics instead of the set semantics, that is, a table can have two copies of the same tuple. The following example illustrates the issue.

Example 7. Since the SQL standard allows duplicate rows, i.e. uses the bag semantics, it is possible to have the database D below. If this database had $P[1]$

P	A	B
	a	b
	a	b

as the primary key, then D would not have been accepted as a consistent state, i.e. the insertion of the second tuple $P(a, b)$ would have been rejected.

This is one of the cases in which the SQL standard deviates from the relational model, where duplicates of a row are not considered. In a commercial DBMS a primary key is checked by adding an index to the primary key and then ensuring that there are no duplicates. Therefore if we try to check the primary key by using the associated functional dependency $P(x, y), P(x, z) \to y = z$ we would not have the same semantics since D satisfies the functional dependency in this classical, first-order representation. □

With the type of first-order constraints that we are considering, we cannot enforce a bag semantics, therefore we will assume that D is consistent.

In order to develop a null-value semantics that goes beyond the ICs supported by DBMSs, we analyze other examples.

Example 8. Consider the UIC $\forall xyzstuw(Person(x, y, z, w) \land Person(z, s, t, u) \to u > w + 15)$, and the database D below. This constraint can be considered

Person	Name	Dad	Mom	Age
	Lee	Rod	Mary	27
	Rod	Joe	Tess	55
	Mary	Adam	Ann	null

as a multi-row check constraint. If we want to naturally extend the semantics for single-row check constraints, D would be consistent iff the condition

evaluates to *true* or *unknown*. In this case, D would be consistent because the condition evaluates to *unknown* for $u = null$ and $w = 27$. Here the relevant attributes to check the IC are *Name, Mom, Age*. □

Example 9. Consider the UIC $\forall xyz(Course(x, y, z) \to Employee(y, z))$ and the database D:

Course	Code	Term	ID
	CS18	W04	34

Employee	Term	ID
	W04	null

Since *Term, ID* is not a primary key of *Employee*, the constraint is not a foreign key constraint, and therefore it is not supported by commercial DBMS. In contrast to foreign key constraints, now we can have a null value in the referenced attributes. In order to extend the semantics used in commercial DBMS. to this case, we refer to the literature. For example, in [25] the satisfaction of this type of constraints is defined as follows: An IC $\forall \bar{x}\bar{y}P(\bar{x}) \to \exists \bar{z}Q(\bar{y}, \bar{z})$ is satisfied if, for every tuple $t_1 \in P$, there exists a tuple $t_2 \in Q$, such that t_1 provides *less or equal information* than t_2, i.e. for every attribute, the value in t_1 is the same as in t_2 or the value in t_1 is *null*. In this example we have the opposite situation: (*W04,34*) does *not* provide less or equal information than (*W04,null*). Therefore, we consider the database to be inconsistent wrt the constraint. Note that the only attributes that are relevant to check the constraint are *Term* and *ID*. □

Examples 6, 5, 8 and 9 show that there are some attributes that are "relevant" when the satisfaction of a constraint is checked against a database.

Definition 2. For t a term, i.e. a variable or a domain constant, let $pos^R(\psi, t)$ be the set of positions in predicate $R \in \mathcal{R}$ where t appears in ψ. The set $\mathcal{A}$ of *relevant attributes* for an IC ψ of the form (1) is
$$\mathcal{A}(\psi) = \{R[i] \mid x \text{ is variable present at least twice in } \psi, \text{ and } i \in pos^R(\psi, x)\} \cup$$
$$\{R[i] \mid c \text{ is a constant in } \psi \text{ and } i \in pos^R(\psi, c)\}. \qquad \square$$

Remember that $R[i]$ denotes a position (or the correspondent attribute) in relation R. In short, the relevant attributes for a constraint are those involved in joins, those appearing both in the antecedent and consequent of (1), and those in φ.

Definition 3. For a set of attributes $\mathcal{A}$ and a predicate $P \in \mathcal{R}$, we denote by $P^{\mathcal{A}}$ the predicate P restricted to the attributes in $\mathcal{A}$. $D^{\mathcal{A}}$ denotes the database D with all its database atoms projected onto the attributes in $\mathcal{A}$, i.e. $D^{\mathcal{A}} = \{P^{\mathcal{A}}(\Pi_{\mathcal{A}}(\bar{t})) \mid P(\bar{t}) \in D\}$, where $\Pi_{\mathcal{A}}(\bar{t})$ is the projection on $\mathcal{A}$ of tuple $\bar{t}$. $D^{\mathcal{A}}$ has the same underlying domain $\mathcal{U}$ as D. $\qquad \square$

Example 10. Consider a UIC $\psi : \forall xyz(P(x,y,z) \to R(x,y))$ and D below.

P	A	B	C
	a	b	a
	b	c	a

R	A	B
	a	5
	a	2

Since x and y appear twice in ψ, $\mathcal{A}(\psi)$ $= \{P[1], R[1], P[2], R[2]\}$. The value in z should not be relevant to check the satisfaction of the constraint, because we only want to make sure that the values in the first two attributes in P also appear in R. Then, checking this is equivalent to checking if $\forall xy(P^{\mathcal{A}(\psi)}(x,y) \to R^{\mathcal{A}(\psi)}(x,y))$ is satisfied by $D^{\mathcal{A}(\psi)}$. For a more complex constraint, such as $\gamma: \forall xyzw(P(x,y,z) \wedge R(z,w) \to \exists v R(x,v) \vee w > 3)$, variable x is relevant to check the implication, z is needed to do the join, and w is needed to check the comparison, therefore $\mathcal{A}(\gamma) = \{P[1], R[1], P[3], R[2]\}$.

$D^{\mathcal{A}(\psi)}$:

$P^{\mathcal{A}(\psi)}$	A	B
	a	b
	b	c

$R^{\mathcal{A}(\psi)}$	A	B
	a	5
	a	2

$D^{\mathcal{A}(\gamma)}$:

$P^{\mathcal{A}(\gamma)}$	A	C
	a	a
	b	a

$R^{\mathcal{A}(\gamma)}$	A	B
	a	5
	a	2

$\qquad \square$

An important observation we can make from Examples 6, 5, 8 and 9 is that, roughly speaking, a constraint is satisfied if any of the relevant attributes has a *null* or the constraint is satisfied in the traditional way (i.e. first-order satisfaction and null values treated as any other constant). We introduce a special predicate $IsNull(\cdot)$, with $IsNull(c)$ true iff c is *null*, instead of using the built-in comparison atom $c = null$, because in traditional DBMS this equality would be always evaluated as *unknown* (as observed in [29], the *unique names assumption* does not apply to null values).

Definition 4. A constraint ψ as in (1) is satisfied in the database instance D, denoted $D \models_N \psi$ iff $D^{\mathcal{A}(\psi)} \models \psi^N$, where ψ^N is
$$\forall \bar{x}(\bigwedge_{i=1}^{m} P_i^{\mathcal{A}(\psi)}(\bar{x}_i) \ \to \ (\bigvee_{v_j \in \mathcal{A}(\psi) \cap \bar{x}} IsNull(v_j) \ \vee \ \exists \bar{z}(\bigvee_{j=1}^{n} Q_j^{\mathcal{A}(\psi)}(\bar{y}_j, \bar{z}_j) \ \vee \ \varphi))), \quad (4)$$
where $\bar{x} = \cup_{i=1}^{m} \bar{x}_i$ and $\bar{z} = \cup_{j=1}^{n} \bar{z}_j$. $D^{\mathcal{A}(\psi)} \models \psi^N$ refers to classical first-order satisfaction where *null* is treated as any other constant in $\mathcal{U}$. $\qquad \square$

We can see from Definition 4 that there are basically two cases for constraint satisfaction: (a) If there is a *null* in any of the relevant attributes in the antecedent, then the constraint is satisfied. (b) If no null values appear in them, then the second disjunct in the consequent of formula (4) has to be checked, i.e, the consequent of the original IC restricted to the relevant attributes. This can be done as usual, treating nulls as any other constant.

Formula (4) is a direct translation of formula (1) that keeps the relevant attributes. In particular, if the original constraint is universal, so is the transformed version. Notice that the transformed constraint is domain independent, and then its satisfaction can be checked by restriction to the active domain.

As mentioned before, the semantics for IC satisfaction introduced in [10] considered that tuples with *null* never generated any inconsistencies, even when the null value was not in a relevant attribute. For example, under the semantics in [10], the instance $\{P(b, null)\}$ would be consistent wrt the IC $\forall xy(P(x, y) \rightarrow R(x))$, but it is intuitively clear that there should be a tuple $R(b)$. The new semantics corrects this, and adjusts to the semantics implemented in commercial DBMS.

Notice that in a database without null values, Definition 4 (so as the definition in [10]) coincides with the traditional, first-order definition of IC satisfaction.

Example 11. Given the ICs: (a) $\forall xyz(P(x, y, z) \rightarrow R(x, y))$, (b) $\forall x(T(x) \rightarrow \exists yzP(x, y, z))$, the database instance D below is consistent.

P	A	B	C
	a	d	e
	b	*null*	g

R	D	E
	a	d

T	F
	b

For (a), the variables x and y are relevant to check the constraint, therefore $\mathcal{A}_1 = \{P[1], R[1], P[2], R[2]\}$; and for (b), the variable x is relevant to check the constraint; therefore $\mathcal{A}_2 = \{P[1], T[1]\}$.

$D^{\mathcal{A}_1}$:

$P^{\mathcal{A}_1}$	A	B
	a	d
	b	*null*

$R^{\mathcal{A}_1}$	D	E
	a	d

$D^{\mathcal{A}_2}$:

$P^{\mathcal{A}_2}$	A
	a
	b

$T^{\mathcal{A}_2}$	F
	b

To check if $D \models_N \forall xyz(P(x, y, z) \rightarrow R(x, y))$, we need to check if $D^{\mathcal{A}_1} \models \forall xy(P^{\mathcal{A}_1}(x, y) \rightarrow (IsNull(x) \vee IsNull(y) \vee R^{\mathcal{A}_1}(x, y)))$ For $x = a$ and $y = d$, $D^{\mathcal{A}_1} \models P^{\mathcal{A}_1}(a, d)$, but none of them is a null value, i.e. $IsNull(a)$ and $IsNull(d)$ are both false, therefore we need to check if $D^{\mathcal{A}_1} \models R^{\mathcal{A}_1}(a, d)$. For $x = b$ and $y = null$, $D^{\mathcal{A}_1} \models P^{\mathcal{A}_1}(b, null)$, and since $D^{\mathcal{A}_1} \models IsNull(null)$, the constraint is satisfied. The same analysis can be done to prove that D satisfies constraint (b), this is by checking $D^{\mathcal{A}_2} \models \forall x(T^{\mathcal{A}_2}(x) \rightarrow (IsNull(x) \vee P^{\mathcal{A}_2}(x)))$

If we add tuple $P(f, d, null)$ to D, it would become inconsistent wrt constraint (a), because $D^{\mathcal{A}_1} \not\models (P^{\mathcal{A}_1}(f, d) \rightarrow (IsNull(f) \vee IsNull(d) \vee R^{\mathcal{A}_1}(f, d)))$.   □

Example 12. Consider the IC ψ: $\forall xywz ((P_1(x, y, w) \wedge P_2(y, z)) \rightarrow \exists u\, Q(x, z, u))$ and the database D:

P_1	A	B	C
	a	b	c
	d	*null*	c
	b	e	*null*
	null	b	b

P_2	D	E
	b	a
	e	c
	d	*null*
	null	b

Q	F	G	H
	a	a	c
	b	*null*	c
	b	c	d
	null	c	a

Variables x, y and z are relevant to check the constraint, therefore the set of relevant attributes is $\mathcal{A}(\psi) = \{P_1[1], P_1[2], P_2[1], P_2[2], Q[1], Q[2]\}$. Then we need to check if $D^{\mathcal{A}(\psi)} \models \forall xyz\,((P_1^{\mathcal{A}(\psi)}(x,y) \wedge P_2^{\mathcal{A}(\psi)}(y,z)) \rightarrow (IsNull(x) \vee IsNull(y) \vee IsNull(z) \vee Q^{\mathcal{A}(\psi)}(x,z))$, where $D^{\mathcal{A}(\psi)}$ is

$P_1^{\mathcal{A}(\psi)}$	A	B
	a	b
	d	$null$
	b	e
	$null$	b

$P_2^{\mathcal{A}(\psi)}$	D	E
	b	a
	e	c
	d	$null$
	$null$	b

$Q^{\mathcal{A}(\psi)}$	F	G
	a	a
	b	$null$
	b	c
	$null$	c

When checking the satisfaction of $D^{\mathcal{A}(\psi)} \models \psi^N$, $null$ is treated as any other constant. For example for $x = d$, $y = null$ and $z = b$, the antecedent of the rule is satisfied since $P_1^{\mathcal{A}(\psi)}(d, null) \in D^{\mathcal{A}}$ and $P_2^{\mathcal{A}(\psi)}(null, b) \in D^{\mathcal{A}}$. If $null$ had been treated as a special constant, with no unique names assumption applied to it, the antecedent would have been false. For these values the consequence is also satisfied, because $IsNull(null)$ is true. In this example, $D^{\mathcal{A}(\psi)} \models \psi^N$, and the database satisfies the constraint. □

Notice that in order for formula (4) to have $\bar{z} \neq \emptyset$, i.e. existential quantifiers, there must exist an atom $Q_j(\bar{y}_j, \bar{z}_j)$ in the corresponding IC of the form (1), such that $\bar{z}_j$ has a repeated variable. This is because that is the only case in which a constraint can have $(\mathcal{A}(\psi) \smallsetminus \bar{x}) \neq \emptyset$.

Example 13. Given $\psi\colon \forall x(P(x,y) \rightarrow \exists z Q(x,z,z))$ and $D = \{P(a,b), P(null,c), Q(a, null, null)\}$, $\mathcal{A}(\psi) = \{P[1], Q[1], Q[2], Q[3]\}$. D satisfies ψ iff $D^{\mathcal{A}} \models \psi^N$, with $D^{\mathcal{A}(\psi)} = \{P^{\mathcal{A}}(a), P^{\mathcal{A}}(null), Q^{\mathcal{A}}(a, null, null)\}$ and $\psi^N\colon \forall x(P^{\mathcal{A}(\psi)}(x) \rightarrow (IsNull(x) \vee \exists z Q^{\mathcal{A}(\psi)}(x,z,z)))$. The constraint is satisfied, because for $x = a$ it is satisfied given that there exists the satisfying value $null$ for z; and for $x = null$ the constraint is satisfied given that $IsNull(null)$ is true. □

The predicate *IsNull* also allows us to specify *NOT NULL*-constraints, which are common in commercial DBMS, and prevent certain attributes from taking a null value. As discussed before, this constraint is different from having $x \neq null$.

Definition 5. A *NOT NULL*-constraint (NNC) is a denial constraint of the form
$$\bar{\forall}\bar{x}(P(\bar{x}) \wedge IsNull(x_i) \rightarrow \mathbf{false}), \tag{5}$$

where $x_i \in \bar{x}$ is in the position of the attribute that cannot take null values. For a NNC ψ, we define $D \models_N \psi$ iff $D \models \psi$ in the classical sense, treating $null$ as any other constant. □

Notice that a NNC is not of the form (1), because it contains the constant $null$. This is why we give a separate definitions for them. By adding NNCs we are able to represent all the constraints of commercial DBMS, i.e. primary keys, foreign key constraints, check constraints and *NOT NULL*-constraints.

Our semantics is a natural extension of the semantics used in commercial DBMSs. Note that: (a) In a DBMS there will never be a join between a null and another value (null or not). (b) Any check constraint with comparison, e.g $<, >, =$, will never create an inconsistency when comparing a null value with any other value. These two features justify our decision in Definition 4 to include

the attributes in the joins and the elements in φ among the attributes that are checked to be null with *IsNull*, because if there is a null in them an inconsistency will never arise.

Our semantics of IC satisfaction with null values allows us to integrate our results in a compatible way with current commercial implementations; in the sense that the database repairs we will introduce later on would be accepted as consistent by current commercial implementations (for the classes of constraints that can be defined and maintained by them).

4 Repairs of Incomplete Databases

Given a database instance D, possibly with null values, that is inconsistent, i.e. D does not satisfy a given set IC of ICs of the kind defined in Section 3 or NNCs. A *repair* of D will be a new instance with the same schema as D that satisfies IC and minimally differs from D.

More formally, for database instances D, D' over the same schema, the *distance* between them was defined in [2] by means of the symmetric difference $\Delta(D, D') = (D \smallsetminus D') \cup (D' \smallsetminus D)$. Correspondingly, a repair of D wrt IC was defined as an instance D' that satisfies IC and minimizes $\Delta(D, D')$ under set inclusion. Finally, a tuple $\bar{t}$ was defined as a consistent answer to a query $Q(\bar{x})$ in D wrt IC if $\bar{t}$ is an answer to $Q(\bar{x})$ from every repair of D wrt IC. The definition of repair given in [2] implicitly ignored the possible presence of null values. Similarly, in [3, 5, 11], that followed the repair semantics in [2], no null values were used in repairs.

Example 14. Consider the database D below and the RIC: $Course(ID, Code) \to \exists Name \ Student(ID, Name)$. D is inconsistent, because there is no tuple in *Student* for tuple *Course(34,C18)* in

Course	ID	Code
	21	C15
	34	C18

Student	ID	Name
	21	Ann
	45	Paul

Course. The database can be minimally repaired by deleting the inconsistent tuple or by inserting a new tuple into table *Student.* In the latter case, since the value for attribute *Name* is unknown, we should consider repairs with all the possible values in the domain. Therefore, for the repair semantics introduced in [2], the repairs are of the two following forms

Course	ID	Code
	21	C15

Student	ID	Name
	21	Ann
	45	Paul

Course	ID	Code
	21	C15
	34	C18

Student	ID	Name
	21	Ann
	45	Paul
	34	μ

for all the possible values of μ in the domain, obtaining a possibly infinite number of repairs.   □

The problem of deciding if a tuple is a consistent answer to a query wrt to a set of universal and referential ICs is undecidable for this repair semantics [11].

An alternative approach is to consider that, in a way, the value μ in Example 14 is an unknown value, and therefore, instead of making it take all the values in the domain, we could use it as a null value. We will pursue this idea, which requires to modify the notion of repair accordingly. It will turn out that consistent query answering will become decidable for universal and referential constraints.

Example 15. (example 14 cont.) By using null values, there will be two repairs:

Repair 1:

Course	ID	Code		Student	ID	Name
	21	C15			21	Ann
					45	Paul

Repair 2:

Course	ID	Code		Student	ID	Name
	21	C15			21	Ann
	34	C18			45	Paul
					34	null

Here *null* tells us that there is a tuple with 34 in the first attribute, but unknown value in the second. □

Now we define in precise terms the notion of repair of a database with null values.

Definition 6. [6] Let D, D', D'' be database instances over the same schema and domain $\mathcal{U}$. It holds $D' \leq_D D''$ iff: (a) For every database atom $P(\bar{a}) \in \Delta(D, D')$, with $\bar{a} \in (\mathcal{U} \smallsetminus \{null\})$,[4] it holds $P(\bar{a}) \in \Delta(D, D'')$; and (b) For every atom $Q(\bar{a}, \overline{null})$[5] $\in \Delta(D, D')$, with $\bar{a} \in (\mathcal{U} \smallsetminus \{null\})$, there exists a $\bar{b} \in \mathcal{U}$ such that $Q(\bar{a}, \bar{b}) \in \Delta(D, D'')$ and $Q(\bar{a}, \bar{b}) \notin \Delta(D, D')$. □

Definition 7. Given a database instance D and a set IC of ICs of the form (1) and NNCs, a repair of D wrt IC is a database instance D' over the same schema, such that $D' \models_N IC$ and D' is $\leq_D$-*minimal* in the class of database instances that satisfy IC wrt $\models_N$, and share the schema with D, i.e. there is no database D'' in this class with $D'' <_D D'$, where $D'' <_D D'$ means $D'' \leq_D D'$ but not $D' \leq_D D''$. The set of repairs of D wrt IC is denoted with $Rep(D, IC)$. □

In the absence of *null*, this definition of repair coincides with the one in [2].

Example 16. The database instance $D = \{Q(a,b), P(a,c)\}$ is inconsistent wrt the ICs $\psi_1 : (P(x,y) \rightarrow \exists z Q(x,z))$ and $\psi_2 : (Q(x,y) \rightarrow y \neq b)$.[6] because $D \not\models_N \psi_2$. The database has two repairs wrt $\{\psi_1, \psi_2\}$, namely $D_1 = \{\}$, with $\Delta(D, D_1) = \{Q(a,b), P(a,c)\}$, and $D_2 = \{P(a,b), Q(a,null)\}$, with $\Delta(D, D_2) = \{Q(a,b), Q(a,null)\}$. Notice that $D_2 \not\leq_D D_1$ because $Q(a,null) \in \Delta(D, D_2)$ and there is no constant $d \in \mathcal{U}$ such that $Q(a,d) \in \Delta(D, D_1)$ and $Q(a,d) \notin \Delta(D, D_2)$. Similarly, $D_1 \not\leq_D D_2$, because $P(a,c) \in \Delta(D, D_1)$ and $P(a,c) \notin \Delta(D, D_1)$. □

Example 17. If the database instance is $\{P(a, null), P(b, c), R(a, b)\}$ and IC consists only of $(P(x,y) \rightarrow \exists z\, R(x,z))$, then there are two repairs: $D_1 = \{P(a, null), P(b, c), R(a, b), R(b, null)\}$, with $\Delta(D, D_1) = \{R(b, null)\}$, and $D_2 = \{P(a, null), R(a, b)\}$, with $\Delta(D, D_2) = \{P(b, c)\}$. Notice, for example, that $D_3 = \{P(a, null), P(b, c), R(a, b), R(b, d)\}$, for any $d \in \mathcal{U}$ different from *null*, is not a repair: Since $\Delta(D, D_3) = \{R(b, d)\}$, we have $D_2 <_D D_3$ and, therefore D_3 is not $\leq_D$-minimal. □

Example 18. Consider the UIC $\forall xy(P(x,y) \rightarrow T(x))$ and the RIC $\forall x(T(x) \rightarrow \exists y P(y, x))$, and the inconsistent database $D = \{P(a, b), P(null, a), T(c)\}$. In this case, we have a RIC-cyclic set of ICs. The four repairs are

---

[4] That $\bar{a} \in (\mathcal{U} \smallsetminus \{null\})$ means that each of the elements in tuple $\bar{a}$ belongs to $(\mathcal{U} \smallsetminus \{null\})$.

[5] $\overline{null}$ is a tuple of null values, that, to simplify the presentation, are placed in the last attributes of Q, but could be anywhere else in Q.

[6] The second IC is *non-generic* [7] in the sense that it implies some ground database literals. Non generic ICs have in general been left aside in the literature on CQA.

i	D_i	$\Delta(D, D_i)$
1	$\{P(a,b), P(null,a), T(c), P(null,c), T(a)\}$	$\{T(a), P(null,c)\}$
2	$\{P(a,b), P(null,a), T(a)\}$	$\{T(a), T(c)\}$
3	$\{P(null,a), T(c), P(null,c)\}$	$\{P(a,b), P(null,c)\}$
4	$\{P(null,a)\}$	$\{P(a,b), T(c)\}$

Notice that, for example, the additional instance $D_5 = \{P(a,b), P(null,a), T(c), P(c,a), T(c)\}$, with $\Delta(D, D_5) = \{T(a), P(c,a)\}$, satisfies IC, but is not a repair because $D_1 <_D D_5$. □

The previous example shows that we obtain a finite number of repairs (with finite extension). If we repaired the database by using the non-null constants in the infinite domain with the repair semantics of [2], we would obtain an infinite number of repairs and infinitely many of them with infinite extension, as considered in [11].

Example 19. Consider a schema with relations $R(X, Y)$, with primary key $R[1]$, and a table $S(U, V)$, with $S[2]$ a foreign key to table R. The ICs are $\forall xyz\ (R(x,y) \wedge R(x,z) \rightarrow y = z)$ and $\forall uv\ (S(u,v) \rightarrow \exists y\ R(v,y))$, plus the NNC $\forall xy(R(x,y) \wedge IsNull(x) \rightarrow \textbf{false})$. Since the original database satisfies the NNC and there is no constraint with an existential quantifier over $R[1]$, the NNC will not be violated while trying to solve other inconsistencies. We would have a *non-conflicting interaction* of RICs and NNCs. Here $D = \{R(a,b), R(a,c), S(e,f), S(null,a)\}$ is inconsistent and its repairs are $D_1 = \{R(a,b), S(e,f), S(null,a), R(f,null)\}$, $D_2 = \{R(a,c), S(e,f), S(null,a), R(f,null)\}$, $D_3 = \{R(a,b), S(null,a)\}$ and $D_4 = \{R(a,c), S(null,a)\}$. □

If a given database D is consistent wrt a set of ICs, then there is only one repair, that coincides with D. The following example shows what can happen if we have a *conflicting interaction* of a RIC containing an existential quantifier over a variable with an additional NNC that prevents that variable from taking null values.

Example 20. Consider the database $D = \{P(a), P(b), Q(b,c)\}$, the RIC $\forall x\ (P(x) \rightarrow \exists y\ Q(x,y))$, and the NNC $\forall xy(Q(x,y) \wedge IsNull(y) \rightarrow \textbf{false})$ over an existentially quantified attribute in the RIC. We cannot repair as expected using null values. Actually, the repairs are $\{P(b), Q(b,c)\}$, corresponding to a tuple deletion, but also those of the form $\{P(a), P(b), Q(b,c), Q(a, \mu)\}$, for every $\mu \in (\mathcal{U} \setminus \{null\})$, that are obtained by tuple insertions. We thus recover the repair semantics of [2]. □

With an appropriate conflicting interaction of RICs and NNCs we could recover in our setting the situation where infinitely many repairs and infinitely many with finite extension appear (c.f. remark after Example 18). Our repair semantics above could be modified in order to repair only through tuple deletions in this case, when null values cannot be used due to the presence of conflicting NNCs. This could be done as follows: If $Rep(D, IC)$ is the class of repairs according to Definitions 6 and 7, the alternative class of repairs, $Rep_d(D, IC)$, that prefers tuple deletions over insertions with arbitrary non-null elements of the domain due to the presence of conflicting NNCs, can be defined by $Rep_d(D, IC) := \{D' \mid D' \in Rep(D, IC)$ and there is no $D'' \in Rep(D, IC')$ with $D'' <_D D'\}$, where IC' is IC without the (conflicting) NNCs.

Since the semantics introduced Definitions 6 and 7 is easier to deal with, and in order to avoid repairs like those in Example 20, we will make the following

Assumption: Our sets IC, consisting of ICs of the form (1) and NNCs, are *non-conflicting*, in the sense that there is no NNC on an attribute that is existentially quantified in an IC of the form (1).

In this way, we will always be able to repair RICs by tuple deletions or tuple insertions with null values. Notice that every set of ICs consisting of primary key constraints (with the keys set to be non-null), foreign key constraints, and check constraints satisfies this condition. Also note that if there are non conflicting NNCs, the original semantics and the one based on Rep_d-repairs coincide. The repair programs introduced in Section 5 compute specify the Rep_d-repairs, so our assumption is also relevant from the computational point of view.

It is possible to prove that under our repair semantics there will always exists a repair for a database D wrt a set of non-conflicting ICs. This follows from the fact that a database instance with no tuples always satisfies the ICs. Furthermore, the set of repairs is finite and each of them is finite in extension (i.e. each database relation is finite). This can be proved establishing first by contradiction that the repairs are restricted to have constants in $adom(D) \cup const(IC) \cup \{null\}$, where $adom(D)$ is the active domain of the original instance D and $const(IC)$ is the set of constants that appear in the ICs. Since the constants that can appear in the repairs are finite there is a finite set of candidates to repairs, and each of them is finite.

Proposition 1. Given a database D and a set IC of non-conflicting ICs: (a) For every repair $D' \in Rep(D, IC)$, $adom(D') \subseteq adom(D) \cup const(IC) \cup \{null\}$. (b) The set $Rep(D, IC)$ of repairs is non-empty and finite; and every $D' \in Rep(D, IC)$ is finite.[7]   □

Theorem 1. The problem of determining if a database D' is a repair of D wrt a set IC consisting of ICs of the form (1) and NNCs[8] is *coNP-complete*.   □

Definition 8. [2] Given a database D, a set of ICs IC, and a query $Q(\bar{x})$, a ground tuple $\bar{t}$ is a *consistent answer* to Q wrt IC in D iff for every $D' \in Rep(D, IC)$, $D' \models Q[\bar{t}]$. If Q is a sentence (boolean query), then *yes* is a consistent answer iff $D' \models Q$ for every $D' \in Rep(D, IC)$. Otherwise, the consistent answer is *no*.   □

In this formulation of CQA we are using a notion $D' \models Q[\bar{t}]$ of satisfaction of queries in a database with null values. At this stage, we are not committing to any particular semantics for query answering in this kind of databases. In the rest of the paper, we will assume that we have such a notion, say $\models_N^q$, that can be applied to queries in databases with null values. Some proposals can be found in the literature [22, 26, 34]. In principle, $\models_N^q$ may be orthogonal to the notion $\models_N$ for satisfaction of ICs. However, in the extended version of this paper we will present a semantics for query answering that is compatible with the one for IC satisfaction. For the moment we are going to assume that $\models_N^q$ can be computed in polynomial time in data for safe first-order queries, and that it coincides

---

[7] For proofs of all results go to www.scs.carleton.ca/~lbravo/IIDBdemos.pdf
[8] In this case we do not need the assumption of non-conflicting ICs

with the classical first-order semantics for queries and databases without null values. We will also assume in the following that queries are safe [33], a sufficient syntactic condition for domain independence.

The decision problem of consistent query answering is

$$CQA(Q, IC) = \{(D, \bar{t}) \mid \bar{t} \text{ is a consistent answer to } Q(\bar{x}) \text{ wrt } IC \text{ in } D\}.$$

Since we have Q and IC as parameters of the problem, we are interested in the data complexity of this problem, i.e. in terms of the size of the database [1]. It turns out that CQA for FOL queries is decidable, in contrast to what happens with the classic repair semantics [2], as established in [11].

Theorem 2. Consistent query answering for first-order queries wrt to non-conflicting sets of ICs of the form (1) and NNCs is decidable. $\square$

The ideas behind the proof are as follows: (a) There is a finite number of database instances that are candidates to be repair given that the use only the active domain of the original instance, *null* and the constants in the ICs. (b) The satisfaction of ICs in the candidates can de decided by restriction to the active domain given that the ICs are domain independent. (c) Checking if $D_1 \leq_D D_2$ can be effectively decided. (d) The answers to safe first-order queries can be effectively computed.

The following proposition can be obtained by using a similar result [15] and the fact that our tuple deletion based repairs are exactly those considered in [15], and every repair in our sense that is not one of those contains at least one tuple insertion.

Theorem 3. Consistent query answering for first-order queries and non-conflicting sets of ICs of the form (1) or NNCs is Π_2^p-complete. $\square$

In the proof of this theorem NNCs are not needed for hardness. Actually, hardness can be obtained with boolean queries.

5 Repair Logic Programs

The stable models semantics was introduced in [18] to give a semantics to disjunctive logic programs that are non-stratified, i.e. that contain recursive definitions that contain weak negation. By now it is the standard semantics for such programs. Under this semantics, a program may have several stable models; and what is true of the program is what is true in all its stable models (a cautious semantics).

Repairs of relational databases can be specified as stable models of disjunctive logic programs. In [6, 10, 12] such programs were presented, but they were based on classic IC satisfaction, that differs from the one introduced in Section 3.

The repair programs we will present now implement the repair semantics introduced in Section 3 for a set of RIC-acyclic constraints. The repair programs use annotation constants with the intended, informal semantics shown in the table below. The annotations are used in an extra attribute introduced in each database predicate; so for a predicate $P \in \mathcal{R}$, the new version of it, $P\_$, contains an extra attribute.

Annotation	Atom	The tuple $P(\bar{a})$ is...
$\mathbf{t_a}$	$P\_(\bar{a}, \mathbf{t_a})$	advised to be made true
$\mathbf{f_a}$	$P\_(\bar{a}, \mathbf{f_a})$	advised to be made false
$\mathbf{t^\star}$	$P\_(\bar{a}, \mathbf{t^\star})$	true or becomes true
$\mathbf{t^{\star\star}}$	$P\_(\bar{a}, \mathbf{t^{\star\star}})$	it is true in the repair

In the repair program, *null* is treated as any other constant in $\mathcal{U}$, and therefore the *IsNull*(x) atom can be replaced by $x = null$.

Definition 9. Given a database instance D, a set *IC* of UICs, RICs and NNCs, the repair program $\Pi(D, IC)$ contains the following rules:

1. Facts: $P(\bar{a})$ for each atom $P(\bar{a}) \in D$.
2. For every UIC ψ of form (2), the rules:
   $$\bigvee_{i=1}^{n} P_{i\text{-}}(\bar{x}_i, \mathbf{f_a}) \vee \bigvee_{j=1}^{m} Q_{j\text{-}}(\bar{y}_j, \mathbf{t_a}) \leftarrow \bigwedge_{i=1}^{n} P_{i\text{-}}(\bar{x}_i, \mathbf{t^\star}), \bigwedge_{Q_{j\text{-}} \in Q'} Q_{j\text{-}}(\bar{y}_j, \mathbf{f_a}),$$
   $$\bigwedge_{Q_k \in Q''} not\ Q_k(\bar{y}_k), \bigwedge_{x_l \in \mathcal{A}(\psi) \cap \bar{x}} x_l \neq null, \bar{\varphi}.$$
   for every set Q' and Q'' of atoms appearing in formula (2) such that $Q' \cup Q'' = \bigcup_{j=1}^{m} Q_j(\bar{y}_j)$ and $Q' \cap Q'' = \emptyset$.[9] Here $\mathcal{A}(\psi)$ is the set of relevant attributes for ψ, $\bar{x} = \bigcup_{i=1}^{n} x_i$ and $\bar{\varphi}$ is a conjunction of built-ins that is equivalent to the negation of φ.
3. For every RIC of form (3), the rules:
   $$P_{\text{-}}(\bar{x}, \mathbf{f_a}) \vee Q_{\text{-}}(\bar{x}', \overline{null}, \mathbf{t_a}) \leftarrow P_{\text{-}}(\bar{x}, \mathbf{t^\star}),\ not\ \ aux(\bar{x}'), \bar{x}' \neq null.$$
   And for every $y_i \in \bar{y}$:
   $$aux(\bar{x}') \leftarrow Q_{\text{-}}(\bar{x}', \bar{y}, \mathbf{t^\star}),\ \ not\ Q_{\text{-}}(\bar{x}', \bar{y}, \mathbf{f_a}),\ \bar{x}' \neq null,\ y_i \neq null.$$
4. For every NNC of the form (5), the rule:
   $$P_{\text{-}}(\bar{x}, \mathbf{f_a}) \leftarrow P_{\text{-}}(\bar{x}, \mathbf{t^\star}), x_i = null.$$
5. For each predicate $P \in R$, the annotation rules:
   $$P_{\text{-}}(\bar{x}, \mathbf{t^\star}) \leftarrow P(\bar{x}). \qquad P_{\text{-}}(\bar{x}, \mathbf{t^\star}) \leftarrow P_{\text{-}}(\bar{x}, \mathbf{t_a}).$$
6. For every predicate $P \in \mathcal{R}$, the interpretation rule:
   $$P_{\text{-}}(\bar{x}, \mathbf{t^{\star\star}}) \leftarrow P_{\text{-}}(\bar{x}, \mathbf{t^\star}),\ not\ P_{\text{-}}(\bar{x}, \mathbf{f_a}).$$
7. For every predicate $P \in \mathcal{R}$, the program denial constraint:
   $$\leftarrow\ \ P_{\text{-}}(\bar{x}, \mathbf{t_a}), P_{\text{-}}(\bar{x}, \mathbf{f_a}). \qquad\qquad\qquad \square$$

Facts in 1. are the elements of the database. Rules 2., 3. and 4. capture, in the right-hand side, the violation of ICs of the forms (2), (3), and (5), resp., and, with the left-hand side, the intended way of restoring consistency. The set of predicates Q' and Q'' are used to check that in all the possible combinations, the consequent of a UIC is not being satisfied. Since the satisfaction of UICs and RICs needs to be checked only if none of the relevant attributes of the antecedent are *null*, we use $x \neq null$ in rule 2. and in the first two rules in 3. (as usual, $\bar{x}' \neq null$ means the conjunction of the atoms $x_j \neq null$ for $x_j \in \bar{x}'$). Notice that rules 3. are implicitly based on the fact that the relevant attributes for a RIC of the form (3) are $\mathcal{A} = \{x \mid x \in \bar{x}'\}$. Rules 5. capture the atoms that are part of the inconsistent database or that become true in the repair process; and rules 6. those that become true in the repairs. Rule 7. enforces, by discarding models, that no atom can be made both true and false in a repair.

Example 21. (example 19 cont.) The repair program $\Pi(D, IC)$ is the following:
1. $R(a, b)$. $R(a, c)$. $S(e, f)$. $S(null, a)$.
2. $R_{\text{-}}(x, y, \mathbf{f_a}) \vee R_{\text{-}}(x, z, \mathbf{f_a}) \leftarrow R_{\text{-}}(x, y, \mathbf{t^\star}), R_{\text{-}}(x, z, \mathbf{t^\star}), y \neq z, x \neq null$.
3. $S_{\text{-}}(u, x, \mathbf{f_a}) \vee R_{\text{-}}(x, null, \mathbf{t_a}) \leftarrow S_{\text{-}}(u, x, \mathbf{t^\star}), not\ aux(x), x \neq null$.
   $aux(x) \leftarrow R_{\text{-}}(x, y, \mathbf{t^\star}),\ \ not\ R_{\text{-}}(x, y, \mathbf{f_a}), x \neq null, y \neq null$.
5. $R_{\text{-}}(x, y, \mathbf{t^\star}) \leftarrow R_{\text{-}}(x, y, \mathbf{t_a})$. $R_{\text{-}}(x, y, \mathbf{t^\star}) \leftarrow R(x, y)$.   (similarly for S)
6. $R_{\text{-}}(x, y, \mathbf{t^{\star\star}}) \leftarrow R_{\text{-}}(x, y, \mathbf{t_a})$.

---

[9] We are assuming in this definition that the rules are a direct translation of the original ICs introduced in Section 2; in particular, the same variables are used and the standardization conditions about their occurrences are respected in the program.

$R_-(x, y, \mathbf{t}^{\star\star}) \leftarrow R(x, y),\ not\ R_-(x, y, \mathbf{f_a}).$ (similarly for S)

7. $\leftarrow R_-(x, y, \mathbf{t_a}), R_-(x, y, \mathbf{f_a}).$ $\leftarrow S_-(\bar{x}, \mathbf{t_a}), S_-(\bar{x}, \mathbf{f_a}).$

Only rules 2. and 3. depend on the ICs: rules 2. for the UIC, and 3. for the RIC. They say how to repair the inconsistencies. In rule 2., $Q' = Q'' = \emptyset$, because there is no database predicate in the consequent of the UIC. There is no rule 4., because there is no NNC. □

Example 22. Consider $D = \{P(a, b), P(c, null)\}$ and the non-conflicting set of ICs: $\{\forall P(x, y) \rightarrow R(x) \vee S(y), P(x, y) \wedge IsNull(y) \rightarrow \mathbf{false}\}$. Then $\Pi(D, IC)$:

1. $P(a, b).$ $P(c, null).$
2. $P_-(x, y, \mathbf{f_a}) \vee R_-(x, \mathbf{t_a}) \vee S_-(y, \mathbf{t_a}) \leftarrow P_-(x, y, \mathbf{t}^\star), R_-(x, \mathbf{f_a}), S_-(y, \mathbf{f_a}), x \neq null, y \neq null.$
   $P_-(x, y, \mathbf{f_a}) \vee R_-(x, \mathbf{t_a}) \vee S_-(y, \mathbf{t_a}) \leftarrow P_-(x, y, \mathbf{t}^\star), R_-(x, \mathbf{f_a}),\ not\ S(y), x \neq null, y \neq null.$
   $P_-(x, y, \mathbf{f_a}) \vee R_-(x, \mathbf{t_a}) \vee S_-(y, \mathbf{t_a}) \leftarrow P_-(x, y, \mathbf{t}^\star),\ not\ R(y), S_-(x, \mathbf{f_a}), x \neq null, y \neq null.$
   $P_-(x, y, \mathbf{f_a}) \vee R_-(x, \mathbf{t_a}) \vee S_-(y, \mathbf{t_a}) \leftarrow P_-(x, y, \mathbf{t}^\star),\ not\ R(y),\ not\ S(y), x \neq null, y \neq null.$
4. $P_-(x, y, \mathbf{f_a}) \leftarrow P_-(x, y, \mathbf{t}^\star), y = null.$
5. $P_-(x, y, \mathbf{t}^\star) \leftarrow P_-(x, y, \mathbf{t_a}).$ $P_-(x, y, \mathbf{t}^\star) \leftarrow P(x, y).$ (similarly for R and S)
6. $P_-(x, y, \mathbf{t}^{\star\star}) \leftarrow P_-(x, y, \mathbf{t_a}).$
   $P_-(x, y, \mathbf{t}^{\star\star}) \leftarrow P(x, y),\ not\ P_-(x, y, \mathbf{f_a}).$ (similarly for R and S)
7. $\leftarrow P_-(x, y, \mathbf{t_a}), P_-(x, y, \mathbf{f_a}).$ (similarly for R and S)

The rules in 2. are constructed by choosing all the possible sets Q' and Q'' such that $Q' \cup Q'' = \{R(x), S(y)\}$ and $Q' \cap Q'' = \emptyset$. The first rule in 2. corresponds to $Q' = \{R(x), S(y)\}$ and $Q'' = \emptyset$, the second for $Q' = \{R(x)\}$ and $Q'' = \{S(y)\}$, the third for $Q' = \{S(y)\}$ and $Q'' = \{R(x)\}$, and the fourth for $Q' = \emptyset$ and $Q'' = \{R(x), S(y)\}$ □

The repair program can be run by a logic programming system that computes the stable models semantics, e.g. DLV system [24]. The repairs can be obtained by collecting the atoms annotated with $\mathbf{t}^{\star\star}$ in the stable models of the program.

Definition 10. Let $\mathcal{M}$ be a stable model of program $\Pi(D, IC)$. The database instance associated with $\mathcal{M}$ is $D_\mathcal{M} = \{P(\bar{a}) \mid P \in \mathcal{R}\ and\ P_-(\bar{a}, \mathbf{t}^{\star\star}) \in \mathcal{M}\}.$ □

Example 23. (example 21 continued) The program has four stable models (the facts of the program are omitted for simplicity):
$\mathcal{M}_1 = \{R_-(a, b, \mathbf{t}^\star),\ R_-(a, c, \mathbf{t}^\star),\ S_-(e, f, \mathbf{t}^\star),\ S_-(null, a, \mathbf{t}^\star),\ aux(a),\ \underline{S_-(e, f, \mathbf{t}^{\star\star})},$
$\underline{S_-(null, a, \mathbf{t}^{\star\star})},\ R_-(f, null, \mathbf{t_a}),\ \underline{R_-(a, b, \mathbf{t}^{\star\star})},\ R_-(a, c, \mathbf{f_a}),\ R_-(f, null, \mathbf{t}^\star),$
$\underline{R_-(f, null, \mathbf{t}^{\star\star})}\ \},$
$\mathcal{M}_2 = \{R_-(a, b, \mathbf{t}^\star),\ R_-(a, c, \mathbf{t}^\star),\ S_-(e, f, \mathbf{t}^\star),\ S_-(null, a, \mathbf{t}^\star),\ aux(a),\ \underline{S_-(e, f, \mathbf{t}^{\star\star})},$
$\underline{S_-(null, a, \mathbf{t}^{\star\star})},\ R_-(f, null, \mathbf{t_a}),\ R_-(a, b, \mathbf{f_a}),\ \underline{R_-(a, c, \mathbf{t}^{\star\star})},\ R_-(f, null, \mathbf{t}^\star),$
$\underline{R_-(f, null, \mathbf{t}^{\star\star})}\ \},$
$\mathcal{M}_3 = \{R_-(a, b, \mathbf{t}^\star),\ R_-(a, c, \mathbf{t}^\star),\ S_-(e, f, \mathbf{t}^\star),\ S_-(null, a, \mathbf{t}^\star),\ aux(a),\ S_-(e, f, \mathbf{f_a}),$
$\underline{S_-(null, a, \mathbf{t}^{\star\star})}, \underline{R_-(a, b, \mathbf{t}^{\star\star})}, R_-(a, c, \mathbf{f_a})\},$
$\mathcal{M}_4 = \{R_-(a, b, \mathbf{t}^\star),\ R_-(a, c, \mathbf{t}^\star),\ S_-(e, f, \mathbf{t}^\star),\ S_-(null, a, \mathbf{t}^\star),\ aux(a),\ S_-(e, f, \mathbf{f_a}),$
$\underline{S_-(null, a, \mathbf{t}^{\star\star})}, R_-(a, b, \mathbf{f_a}), \underline{R_-(a, c, \mathbf{t}^{\star\star})}\}.$

The databases associated to the models select the underlined atoms: $D_1 = \{S(e, f), S(null, a), R(a, b), R(f, null)\}$, $D_2 = \{S(e, f), S(null, a), R(a, c), R(f, null)\}$ $D_3 = \{S(null, a), R(a, b)\}$ and $D_4 = \{S(null, a), R(a, c)\}$. As expected these are the repairs obtained in Example 19. □

Theorem 4. Let IC be a RIC-acyclic set of UICs, RICs and NNCs. If $\mathcal{M}$ is a stable model of $\Pi(D, IC)$, then $D_\mathcal{M}$ is a repair of D with respect to IC. Furthermore, the repairs obtained in this way are all the repairs of D. □

6 Head-Cycle-Free Programs

In some cases, the repair programs introduced in Section 5 can be transformed into equivalent non-disjunctive programs. This is the case when they become *head-cycle-free* [8]. Query evaluation from such programs has lower computational complexity than general disjunctive programs, actually the data complexity is reduced from Π_2^P-complete to *coNP*-complete [8, 16]. We briefly recall their definition.

The *dependency graph* of a ground disjunctive program Π is the directed graph that has ground atoms as vertices, and an edge from atom A to atom B iff there is a rule with A (positive) in the body and B (positive) in the head. Π is *head-cycle free* (HCF) iff its dependency graph does not contain any directed cycles passing through two atoms in the head of the same rule. A disjunctive program Π is HCF if its ground version is HCF.

A HCF program Π can be transformed into a non-disjunctive normal program $sh(\Pi)$ that has the same stable models. It is obtained by replacing every disjunctive rule of the form $\quad \bigvee_{i=1}^{n} P_i(\bar{x}_i) \leftarrow \bigwedge_{j=1}^{m} Q_j(\bar{y}_j), \varphi.\quad$ by the n rules $P_i(\bar{x}_i) \leftarrow \bigwedge_{j=1}^{m} Q_j(\bar{y}_j), \varphi, \bigwedge_{k \neq i} not\ P_k(\bar{x}_k).$, for $i = 1, ..., n$.

For certain classes of queries and ICs, consistent query answering has a data complexity lower than Π_2^P, a sharp lower bound as seen in Theorem 3 (c.f. also [15]). In those cases, it is natural to consider this kind of transformations of the disjunctive repair program. In the rest of this section we will consider sets IC of integrity constraints formed by UICs, RICs and NNCs.

Definition 11. A predicate P is *bilateral* with respect to IC if it belongs to the antecedent of a constraint $ic_1 \in IC$ and to the consequent of a constraint $ic_2 \in IC$, where ic_1 and ic_2 are not necessarily different. □

Example 24. If $IC = \{\forall x\ (T(x) \rightarrow \exists y\ R(x,y), \forall xy\ (S(x,y) \rightarrow T(x))\}$, the only bilateral predicate is T. □

Theorem 5. For a set IC of UICs, RICs and NNCs, if for every $ic \in IC$, it holds that (a) ic has no bilateral predicates; or (b) ic has exactly one occurrence of a bilateral predicate (without repetitions), then the program $\Pi(D, IC)$ is HCF. □

For example, if in IC we have the constraint $P(x,y) \rightarrow P(y,x)$, then P is a bilateral predicate, and the condition in the theorem is not satisfied. Actually, the program $\Pi(D, IC)$ is not HCF. If we have instead $P(x,a) \rightarrow P(x,b)$, even though the condition is not satisfied, the program is HCF. Therefore, the condition is sufficient, but not necessary for the program to be HCF.

This theorem can be immediately applied to useful classes of ICs, like denial constraints, because they do not have any bilateral literals, and in consequence, the repair program is HCF.

Corollary 1. If IC contains only constraints of the form $\bar{\forall}(\bigwedge_{i=1}^{n} P_i(\bar{t}_i) \rightarrow \varphi)$, where $P_i(\bar{t}_i)$ is a database atom and φ is a formula containing built-in predicates only, then $\Pi(D, IC)$ is HCF. □

As a consequence of this corollary we obtain, for first-order queries and this class of ICs, that CQA belongs to *coNP*, because a query program (that is non-disjunctive) together with the repair program is still HCF. For this class of constraints, with the classical tuple-deletion based semantics, this problem becomes *coNP*-complete [15]. Actually, CQA for this class with our tuple-deletion/null-value based semantics is still *coNP*-complete, because the same reduction found in [15] can be used in our case.

7 Conclusions

We have introduced a new repair semantics that considers, systematically and for the first time, the possible occurrence of null values in a database in the form we find them present and treated in current commercial implementations. Null values of the same kind are also used to restore the consistency of the database. The new semantics applies to a wide class of ICs, including cyclic sets of referential ICs.

We established the decidability of CQA under this semantics, and a tight lower and upper bound was presented. The repairs under this semantics can be specified as stable models of a disjunctive logic program with a stable model semantics for acyclic foreign key constraints, universal ICs and *NOT NULL*-constraints, covering all the usual ICs found in database practice.

In an extended version of this paper we will provide: (a) An extension of our semantics of IC satisfaction in databases with null values that can also be applied to query answering in the same kind of databases. (b) A more detailed analysis of the way null-values are propagated in a controlled manner, in such a way that no infinite loops are created. (c) Construction of repairs based on a sequence of "local" repairs for the individual ICs.

References

[1] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Arenas, M., Bertossi, L. and Chomicki, J. Consistent Query Answers in Inconsistent Databases. In *Proc. ACM Symposium on Principles of Database Systems (PODS 99)*, ACM Press, 1999, pp. 68-79.

[3] Arenas, M., Bertossi, L. and Chomicki, J. Answer Sets for Consistent Query Answers. *Theory and Practice of Logic Programming*, 2003, 3(4-5):393-424.

[4] Atzeni, P. and Morfuni, N. Functional Dependencies and Constraints on Null Values in Database Relations. *Information and Control*, 1986, 70(1):1-31.

[5] Barcelo, P. and Bertossi, L. Logic Programs for Querying Inconsistent Databases. In *Proc. Fifth International Symposium on Practical Aspects of Declarative Languages (PADL 03)*, Springer LNCS 2562, 2003, pp. 208-222.

[6] Barcelo, P., Bertossi, L. and Bravo, L. Characterizing and Computing Semantically Correct Answers from Databases with Annotated Logic and Answer Sets. In *Semantics of Databases*, Springer LNCS 2582, 2003, pp. 1-27.

[7] Bertossi, L. and Chomicki, J. Query Answering in Inconsistent Databases. In *Logics for Emerging Applications of Databases*. Springer, 2003, pp. 43-83.

[8] Ben-Eliyahu, R. and Dechter, R. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics in Artificial Intelligence*, 1994, 12:53-87.

[9] Bertossi, L. and Bravo, L. Consistent Query Answers in Virtual Data Integration Systems. In *Inconsistency Tolerance*, Springer LNCS 3300, State of the Art Survey Series, 2004, pp. 42-83.

[10] Bravo, L. and Bertossi, L. Consistent Query Answering under Inclusion Dependencies. In *Proc. Annual IBM Centers for Advanced Studies Conference (CASCON 04)*, 2004, pp. 202-216.

[11] Cali, A., Lembo, D. and Rosati, R. On the Decidability and Complexity of Query Answering over Inconsistent and Incomplete Databases. In *Proc. ACM Symposium on Principles of Database Systems (PODS 03)*, ACM Press, 2003, pp. 260-271.

[12] Caniupan, M. and Bertossi, L. Optimizing Repair Programs for Consistent Query Answering. In *Proc. International Conference of the Chilean Computer Science Society (SCCC 05)*, IEEE Computer Society Press, 2005, pp. 3-12.

[13] Celle, A. and Bertossi, L. Querying Inconsistent Databases: Algorithms and Implementation. In *Computational Logic* , Springer LNCS 1861, 2000, pp. 942-956.

[14] Chomicki, J., Marcinkowski, J. and Staworko, S. Computing Consistent Query Answers Using Conflict Hypergraphs. In *Proc. ACM International Conference on Information and Knowledge Management*, ACM Press, 2004, pp. 417-426.

[15] Chomicki, J. and Marcinkowski, J. Minimal-Change Integrity Maintenance using Tuple Deletions. *Information and Computation*, 2005, 197(1-2):90-121.

[16] Dantsin, E., Eiter, T., Gottlob, G. and Voronkov, A. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 2001, 33(3): 374-425.

[17] Fuxman, A., Fazli, E. and Miller, R.J. ConQuer: Efficient Management of Inconsistent Databases. In *Proc. ACM International Conference on Management of Data (SIGMOD 05)*, ACM Press, 2005, pp. 155-166.

[18] Gelfond, M. and Lifschitz, V. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 1991, 9:365–385.

[19] Grahne, G. *The Problem of Incomplete Information in Relational Databases*. Springer LNCS 554, 1991.

[20] Greco, G., Greco, S. and Zumpano, E. A Logical Framework for Querying and Repairing Inconsistent Databases. *IEEE Transactions in Knowledge and Data Engineering*, 2003, 15(6):1389-1408.

[21] Imielinski, T. and Lipski, W. Incomplete Information in Relational Databases. *Journal of the ACM*, 1984, 31(4):761-791.

[22] International Organization for Standardization (ISO). *ISO International Standard: Database Language SQL - Part 2: SQL/Foundation*, Melton, J. (ed), ISO/IEC 9075-2:2003, 2003.

[23] Lenzerini, M. Data Integration: A Theoretical Perspective. In *Proc. ACM Symposium on Principles of Database Systems (PODS 02)*, ACM Press, 2002, pp. 233-246.

[24] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. The DLV System for Knowledge Representation and Reasoning. To appear in *ACM Transactions on Computational Logic.*

[25] Levene, M. and Loizou, G. Null Inclusion Dependencies in Relational Databases. *Information and Computation*, 1997, 136(2):67–108.

[26] Levene, M. and Loizou, G. *A Guided Tour of Relational Databases and Beyond*, Springer-Verlag, 1999.

[27] Libkin, L. A Semantics-based Approach to Design of Query Languages for Partial Information. In *Semantics in Databases*, Springer LNCS 1358, 1998, pp. 170-208.

[28] Lien, E. On the Equivalence of Database Models. *Journal of the ACM*, 1982, 29(2):333–362.

[29] Reiter, R. Towards a Logical Reconstruction of Relational Database Theory. In *On Conceptual Modelling*, M.L. Brodie, J. Mylopoulos, J.W. Schmidt (eds.), Springer, 1984, pp. 191-233.

[30] Reiter, R. A Sound and Sometimes Complete Query Evaluation Algorithm for Relational Databases with Null Values. *Journal of the ACM*, 1986, 33(2):349-370.

[31] Türker, C. and Gertz, M. Semantic Integrity Support in SQL:1999 and Commercial (Object–) Relational Database Management Systems. *The VLDB Journal*, 2001, 10(4):241-269.

[32] Van der Meyden, R. Logical Approaches to Incomplete Information: A Survey. In *Logics for Databases and Information Systems*, Kluwer, 1998, pp. 307-356.

[33] Van Gelder, A. and Topor, R. Safety and Correct Translation of Relational Calculus Formulas. In *Proc. ACM Symposium on Principles of Database Systems (PODS 87)*, ACM Press, 1987, pp. 313–327.

[34] Zaniolo, C. Database Relation with Null Values. In *Journal of Computer and System Sciences*, 1984, 28(1):142-166.